Lecture 17: Analysis Wrap-up, More Lambdas
10:00 AM, Oct 11, 2019

Contents

1 Analysis ................................. 1
   1.1 Complete Analyses .................. 2

2 Set Equality ............................. 3

3 Lambdas .................................. 4

4 Summary .................................. 4

1 Analysis

We are going to revisit big O. We have been dealing with big-O Functions that take natural numbers and produce real numbers. be warned that you might take a math class that goes reals to reals. Operation counting functions are always this kind because we never do 0 operations. We say ” f is eventually less than g, up to constants” if there are numbers M, C 0 with the property that for n 0, we have

\[ f(n) \leq c \cdot g(n) \]

The notation O(g) denotes the set of all functions that are eventually less than g, up to constants.

Now we have our first big-O theorem. Think of big-O as a function that consumes an operation counting function \( g(n \mapsto r) \) big-O of g is a collection of functions. If A,B 0, and the function

\( H: \mathbb{N} \mapsto \mathbb{R} \) satisfies a recurrence where \( H(0) = A \) and \( H(n) \leq B + H(n - 1) \) for \( n \geq 0 \) then \( H \in O(n \mapsto n) \). This also works if \( H(n) \leq B + H(n - 1) \) for \( n \geq 0 \).

The second big-O theorem says If A,B,C 0, and the function \( H: \mathbb{N} \mapsto \mathbb{R} \) satisfies a recurrence where \( H(0) = A \) and \( H(n) \leq B + Cn + H(n - 1) \) for \( n \geq 0 \) then \( H \in O(n \mapsto n^2) \). This also works if \( H(n) < B + Cn + H(n - 1) \) for \( n \geq 0 \).

Why this obsession with big-O? Well because it is what computer scientists use to speak about algorithms. For example saying merge sort is in \( O(n \mapsto n\log n) \) is shorthand for ”the operation-counting function for the merge sort algorithm is an element of the set \( O(n \mapsto n\log n) \)”

If one function is contained in another then so is their respective big(O) categories. Give an example of a function that lies in \( O(n \mapsto 1) \). Remember a function is always less than or equal to itself. So the function itself is in this set. \( O(n \mapsto 0) \) is also in there. \( 1/n + 1 \) and \( \cos(n) \) are all in there. Think of them as constant time functions. Constant time functions sit inside a bigger circle of linear time functions. \( O(n \mapsto n) \). There is an even bigger category that encompasses both of them, \( O(n \mapsto n^2) \), that is quadratic time. Bigger classes are are \( O(n \mapsto 2^n) \) and even worse is \( O(n \mapsto n^n) \) It is harder to write linear time procedure than quadratic time ones. So we say the
smaller set a function is in. People will be more impressed by ”I’ve got a linear time algorithm!”
than ”I’ve got a quadratic-time algorithm”.

From now on, I’m constantly going to be asking ”Is what we’ve written here linear-time? Constant
time? Quadratic time?” You’ll get better at answering with experience.

There are two more small but very useful theorems. Big-O theorem 3 which says if \(H : N \rightarrow R\),
and \(H \in O(n \mapsto n)\), then there are numbers A,B \(\in R\) with the property that for all natural numbers
\(n\), we have:

\[H(n) < An + B\]

This isn’t obvious because we know that for some M, c \(\in R\) having \(n \leq M\) tells us \(H(n) \leq cn\), so can’t
we just choose \(A = c\), \(B = 0\)? We could but that only make the inequality true for \(n \leq M\), not for all \(n\).

Spike walked through an example in class where \(H(n) = 10n + 1 + n\). He showed the \(H(n)\) is slightly
larger than \(K(n) = n\) by graphing. We want \(K(n)\) to eventually be larger than \(H(n)\). We don’t have
to show is less than \(n \mapsto n\). We have to show it is less than some multiple of \(n\). Try \(2n\) and we see
\(H(n)\) is clearly eventually less than that as shown in class and on the slides. We want some linear
function that is bigger than \(H(n)\) for every \(n\), \(2n\) is not it because for \(0, 1, 2\) this function is less
than \(H(n)\). How can I fix this? What if I took this function and moved it up, say 10 units, then it
would be above \(H(n)\) always. To get this 10, I looked at where \(H(n)\) was still bigger so \(0, 1, 2\) \(n\) values
and the biggest of those values was 10.

Now we are going to do a proof for Theorem 3. If \(H : N \rightarrow R\), and \(H \in O(n \mapsto n)\), then there are
numbers A,B \(\in R\) with the property that for all natural numbers \(n\), we have \(H(n) < An + B\)

Proof:

1. The definition of big-O tells us that there are numbers M, c \(\in R\) with the property that for \(n \leq M\),
   \(M \leq 0\), we have \(H(n) \leq cn\), hence \(H(n) < (c + 1)n\). Let \(B = c + 1\).

2. Look at the numbers \(H(0), H(1), ..., H(k)\), where \(k\) is the first integer greater than \(M\).
   Let \(A\) be one more than the largest of these numbers. Then for \(n = 0, 1, ..., k\), we have
   \(H(n) < A \leq A + Bn\)

3. By step 1, for \(n \leq k\), we have \(H(n) < Bn \leq A + Bn\). So for all natural numbers \(n\), we have \(H(n) \leq A + Bn\), as claimed!

Big-O theorem 4 states that if \(H : N \mapsto R\), and \(H \in O(n \mapsto n^2)\), then there are numbers A,B \(\in R\)
with the property that for all natural numbers \(n\), we have \(H(n) < An^2 + B\) The proof is essentially
the same as for theorem 3.

1.1 Complete Analyses

Remember the function contains17? We are going to show it runs in \(O(n \mapsto n)\) time. Here is the
code.

```
(define (contains17? aloi)
  (cond
    [(empty? aloi) false]
    [(cons? aloi) (or (= (first aloi) 17) (contains17? (rest aloi)))])
```
Let $C(n)$ be the largest number of operations involved in evaluating contains? on any list of length $n$. Then $C : \mathbb{N} \rightarrow \mathbb{R}$, because the number of operations is always positive. From lines 1, 2, 3 of code, we see that $C(0)$ is some small constant $A \leq 0$ as cond, empty?, testing a boolean, etc., are all unit-time operations. From lines 1-4, we can see that for list of nonzero length $n$, the number of operations aside from those in the recursive call in line 4, is some other small constant, for similar reasons; the time taken in the recursive call is no more than $C(n-1)$, because the recursive argument has length $n - 1$, and $C(n - 1)$ represents the greatest possible number of operations on such a list. Big-O theorem 1 tells us that $C \in O(n \rightarrow n)$. QED

Essentially the same argument lets us conclude that Member? also runs in linear time. So now we can analyze set?

```
(define (set? aloi)
  (cond
    [(empty? aloi) true]
    [(cons? aloi) (and (not (member? (first aloi) (rest aloi)))
                         (set? (rest aloi)))]))
```

Let $S$ be the largest number of operations involved in evaluating set? on any list of length $n$. Then $S : \mathbb{N} \rightarrow \mathbb{R}$ because the number of operations is always positive. From lines 1, 2, 3 of code, we see that $S(0)$ is some small constant $A \leq 0$ as cond, empty?, testing a boolean, etc., are all unit-time operations. From lines 1-5, we can see that for list of nonzero length $n$, the number of operations aside from those in the recursive call in line 4, is some other small constant, for similar reasons, plus the time taken by member? on a list of length $n-1$. The time taken in the recursive call is no more than $S(n-1)$, because the recursive argument has length $n - 1$, and $S(n-1)$ represents the greatest possible number of operations on such a list. Big-O theorem 3 tells us there are numbers $C,D \leq 0$ such that $M(n) < Cn + D$ for all $n$, so $M(n - 1) < C(n-1) + D$.

$$S(n) < B + C(n - 1) + D + S(n - 1) = (B + D - C) + Cn + S(n - 1)$$

Note that $(B+D)$ and $C$ are both positive, so big-O theorem 2 applies! Applying big O theorem 2, we see that $S \in O(n \rightarrow n^2)$ QED

That is it for analysis for now. It is always going to look like this, defining some op-counting function and writing a recurrence relation.

## 2 Set Equality

For the homework this week we are going to be representing sets as list of numbers, order doesn’t matter. Ordinary list-equality isn’t a test of set-equality. (list 1 2) (list 2 1) are different lists but the same set. We have to write a procedure to test set equality. Set B and C are equal if every member of B is a member of C and every member of C is a member of B. So they have to be subsets of each other.

```
(define (set-equal? b c) (and (subset? b c) (subset? c b)))
```

Now we have to write subset? For b to be a subset of c, each element of b has to be an element of c.

```
(define (subset? b c)
  (foldr (lambda (item result) (and (member? item c) result))
         true b))
```
The is a subtlety. You have to be able to test equality between items. We can do it for a lot of things like nums, bools, strings and so on. But it is impossible to test equality of procedures. So for homework we restricted to atomic types so you can’t have lists of procedures.

## 3 Lambdas

Lambdas are pretty cool, could I define everything we’ve done so far with lambda? No, we can’t write a recursive procedure using lambda.

```scheme
(let
  ((len (lambda (alod)
                (cond
                 [(empty? alod) 0]
                 [(cons? alod) (+ 1 (len (rest alod)))])))
   (len (list 1 2 3)))
```

The inner `len` cannot refer to the outer one by the rules of evaluation. A solution is to use `letrec`.

```scheme
(letrec
  ((len (lambda (alod)
                (cond
                 [(empty? alod) 0]
                 [(cons? alod) (+ 1 (len (rest alod)))])))
   (len (list 1 2 3))) => 3
```

There is another secret about lambda. We said the value of a lambda expression was a "closure" containing an argument list and the body. The truth is there is a third part of the closure. It also contains a "local environment", informally consisting of all the "bindings" in the current environment except those in the top-level-environment. Thus, the rule for evaluating lambda expression changes too:

1. First extend the current environment by the closure’s local environment
2. bind formal arguments to actual arguments
3. evaluate the body in the extended environment
4. remove the extensions from step 1.

Here is an example:

```scheme
(define (incrementer b) (lambda (x) (+ x b))
(define (add2 x) (incrementer 2))
(add2 5)
```

Without the "local environment", we’d get "b undefined" error! this will be, for many of you, the single hardest part of the next project.
4 Summary

Ideas

- Testing set equality
- Big-O runtime theorems
- Local environments with lambdas

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback